Paradigmi di Programmazione - A.A. 2021-22

Esame Scritto del 15/12/2021

CRITERI DI VALUTAZIONE:
La prova è superata se si ottengono almeno 12 punti negli esercizi 1,2,3 e almeno 18 punti complessivamente.

Esercizio 1 [Punti 4]

Applicare la β-riduzione alla seguente λ-espressione fino a raggiungere una espressione non ulteriormente riducibile o ad accorgersi che la derivazione è infinita:

$$(\lambda x.\lambda y.((\lambda y.yx)(\lambda x.xy)))(xy)$$

Nella soluzione, mostrare tutti i passi di riduzione calcolati, sottolineando ad ogni passo la porzione di espressione a cui si applica la β-riduzione (redex) ed evidenziando le eventuali α-conversioni.

SOLUZIONE:

Una possibile soluzione. La prima operazione è una α-conversione che rende distinti tutti i legatori λ delle variabili presenti ne"espressione originale. Si ottiene pertanto la λ-espressione

$$(\lambda x_1.\lambda y_1.(\lambda y_2.y_2x_1)(\lambda x_2.x_2y_1))(xy)$$

A questo punto applichiamo al β-riduzione

$$\begin{array}{ll} & \underline{(\lambda x_1.\lambda y_1.((\lambda y_2.y_2x_1)(\lambda x_2.x_2y_1)))(xy)} \\ \rightarrow & \lambda y_1.\underline{((\lambda y_2.y_2(xy))(\lambda x_2.x_2y_1))} \\ \rightarrow & \lambda y_1.\underline{((\lambda x_2.x_2y_1)(xy))} \\ & \lambda y_1.(x,y)y_1 \end{array}$$

Esercizio 2 [Punti 4]

Indicare il tipo della seguente funzione OCaml, mostrando i passi fatti per inferirlo:

```
let f x y =
  match x with
  | [] -> y 0
  | z::z' -> y z;;
```

SOLUZIONE:
Struttura del tipo:

```
X -> Y -> RIS
```

Uso per convenzione `X,Y,Z` come variabili di tipo per le variabili `x,y,z`, `RIS` come variabile di tipo del risultato, e `A,B,C,...` come variabili di tipo "fresche" per la definizione dei vincoli.

Vincoli:

```
X = A list      (da pattern matching)
Y = int -> B    (da (y 0) nel primo caso del p.m.)
Z = int         (da (y z) nel secondo caso del p.m.)
Z = A           (da pattern z::z' nel p.m.)
RIS = B         (da (z y) + 1 nel secondo caso del p.m.)
```

Ne consegue:

```
X = int list
Y = int -> B
RIS = B
```

Tipo inferito:

```
int list -> (int -> B) -> B
```

che in sintassi OCaml corrisponde a:

```
(int list) -> (int -> 'a ) -> 'a
```

Esercizio 3 [Punti 7]

Un multi-insieme (multiset) è un insieme che può contenere più istanze dello stesso elemento (il numero delle istanze di un elemento è chiamato "molteplicità" di quell'elemento). Una possibile rappresentazione di un multiset è tramite una lista di valori, esprimibile tramite la seguente definizione di tipo `mset` (polimorfo):

```
type 'a mset_lst = 'a list
```

Una rappresentazione alternativa, più efficiente in termini di occupazione di memoria, è tramite una lista di coppie

```
type 'a mset_map = ('a * int) list
```

in cui ogni coppia `(x,n)` indica che il multiset contiene `n` istanze dell'elemento `x`. Questa rappresentazione si basa sull'assunzione che non ci siano due coppie che contengono lo stesso elemento `x`, e che in tutte le coppie la molteplicità `n` sia maggiore di 0.

Ad esempio, il multiset $\{2, 4, 3, 2, 2, 3\}$ può essere rappresentato come `[2;4;3;2;2;3]` di tipo `int mset_lst` oppure come `[(2,2);(4,1);(3;2)]` di tipo `int mset_map`.

Definire, usando i costrutti di programmazione funzionale in OCaml, le seguenti funzioni:

1. `mult m e` di tipo `'a mset_lst -> 'a -> int` che calcola la molteplicità di `e` nel multiset `m` rappresentato come lista semplice
2. `mult m e` di tipo `'a mset_map -> 'a -> int` che calcola la molteplicità di `e` nel multiset `m` rappresentato come lista di coppie

3. `sum m1 m2` di tipo `'a mset_map -> 'a mset_map -> 'a mset_map` che calcola l'unione dei multiset `m1` ed `m2` rappresentati come liste di coppie (esempio: `sum [('a',2);('b',3)] [('b',4);('c',1)]`) restituisce `[('a',2);('b',7);('c',1)]` (eventualemnte anche con le coppie in ordine diverso)

SOLUZIONE:

Una possibile soluzione (rivedere per sistemare i dettagli modificati...):

1.
```
let rec mult (s : 'a mset_lst) (e : 'a) : int =
            match s with
              [] -> 0
            | elt::ss-> if el = e then 1+mult(ss, e)) else (mult ss e)
```
2.
```
let rec mult (xs : 'a mset_map) (e : 'a) : int =
            match xs with
              [] -> 0
            | (el, ct)::t -> if el = e then ct else (mult t e)
```
3.
```
let rec sum (a : 'a mset_map) (b : 'a mset_map) : ('a mset_map) =
      fold (fun acc (k, _) ->
            if (mult acc k) <> 0 then acc
            else (k, (mult a k) + (mult b k))::acc)
            [] (a @ b)
```

Esercizio 4 [Punti 15]

Si estenda il linguaggio MiniCaml con una nuovo costrutto `unless` che consente di definire espressioni condizionali aventi la seguente forma:

```
e1 unless e2
```

Intuitivamente, in una espressione `e1 unless e2` la valutazione di `e1` è condizionata alla valutazione dell'espressione `e2`. Se la valutazione di `e2` restituisce **false**, allora il risultato della valutazione di `e1 unless e2` sarà dato dalla valutazione di `e1`. Altrimenti, se la valutazione di `e2` restituisce **true**, il risultato della valutazione di `e1 unless e2` sarà il valore **Error**.

Esempi (in sintassi concreta):

```
(3+2) unless (7<0)       (* risultato: 5 *)
(3+2) unless (7>0)       (* risultato: Error *)
(x/y) unless (y=0)       (* controlla il denominatore prima di fare la divisione *)
```

Il valore **Error** si propaga, facendo fallire l'intera espressione che lo contiene (ossia, l'intera espressione restituirà **Error**). Ad esempio, l'intera espressione:

```
let x=((2*y) unless (y=0)) in x+1
```

restituisce come risultato il valore di `2y+1` se `y<>0`, mentre restituisce `Error` se `y=0`. Analogamente, le espressioni:

```
((fun x -> x/y) unless (y=0))(100)
```

```
(fun x-> x/y) (100 unless (y=0))
```

restituiscono entrambe come risultato il valore di `100/y` se `y<>0`, mentre restituisce `Error` se `y=0`.

Si modifichi l'implementazione dell'interprete del linguaggio con quanto serve per gestire il nuovo costrutto proposto, e si fornisca una nuova versione dell'implementazione dei costrutti `Apply ()` (solo per funzioni non ricorsive) e `Let` che tenga conto della propagazione del valore `Error` come negli esempi riportati sopra.

SOLUZIONE:
Una possibile soluzione:

```
type exp = ...
         | Unless of exp * exp

type evT = ...
         | Error

let rec eval e env =
  match e with
    ...
  | Unless (e1,e2) ->
      (match (eval e2 env) with
      | Bool true  -> Error
      | Bool false -> eval e1 env
      | _ -> failwith "tipo non corretto")


  | Apply (e1,e2) -> l
        (match eval e1 env in
        | Error -> Error
        | Closure(x, body, decEnv) ->
                let val = eval e2 env in
                  (match val with
                        | Error -> Erroe
                        | _ -> let env' = bind decEnv x val in
                                                  eval body env')
         | _ -> failwith "Errore di tipo")

  | Let (x,e1,e2) -> let val = eval e1 env in
                     if val=Error then Error
                     else let env' = bind env x val in
                     eval e2 env'
```